The Impact of MPI Queue Usage on Message Latency

Keith D. Underwood Ron Brightwell

Sandia National Laboratories*
P.O. Box 5800 MS-1110
Albuquerque, NM 87185-1110
E-mail: {kdunder,rbbrigh}@sandia.gov

Abstract

It is well known that traditional micro-benchmarks do not fully capture the salient architectural features that impact application performance. Even worse, micro-benchmarks that target MPI and the communications subsystem do not accurately represent the way that applications use MPI. For example, traditional MPI latency benchmarks time a ping-pong communication with one send and one receive on each of two nodes. The time to post the receive is never counted as part of the latency. This scenario is not even marginally representative of most applications. Two new micro-benchmarks are presented here that analyze network latency in a way that more realistically represents the way that MPI is typically used. These benchmarks are used to evaluate modern high-performance networks, including Quadrics, InfiniBand, and Myrinet.

1 Introduction

A significant challenge in the assessment of parallel computers is the poor correlation of micro-benchmarks and the way that applications use the system. This is particularly prevalent in benchmarks that assess the performance of the network subsystem and the MPI library. A particularly egregious example of this is a standard ping-pong latency benchmark. in which one node sends a single message to another. After receiving the message, the second node sends a reply. This is repeated several times to obtain an average latency. On both nodes, the receive is posted outside of the time measurements.

Recent work[6] indicates that applications deviate from this behavior in two significant ways: they have numerous receives that are pre-posted and many unexpected messages[1]. Increasing the number of posted receives increases the amount of work that must be performed when a message is received because the list of posted receives must be searched to find a matching receive. In turn, unexpected messages increase the amount of work that must be done when a receive is posted. Standard MPI micro-benchmarks do not measure either scenario. Indeed, these micro-benchmarks do not even measure the time required to perform an `MPI_Irecv` or `MPI_Recv` as part of the latency path. This fails to represent real applications in what is measured (receives are a part of each computation/communication phase) and in how MPI queues are used.

This paper presents two new micro-benchmarks that analyze the behavior of an MPI implementation over a given network in the presence of longer posted receive queues and longer unexpected message queues. These micro-benchmarks are then used to analyze four networks: Quadrics[16], Myrinet[4], Infiniband[9], and the custom network on ASCI Red[20]. Each of these systems takes a slightly different approach to the integration of the network with the node and a slightly different approach to MPI offloading. The results show that both the weight of the protocol and the speed of the processor have significant impacts on message latency when the queues are heavily utilized. This suggests that more processing capability needs to be allocated to MPI processing — particularly for networks that offload a portion of MPI.

The remainder of this paper is organized as follows. Section 2 presents related work. Section 3 describes the benchmarks used while Section 4 covers the experimental platforms. Results are presented in Section 5 followed by conclusions in Section 6 and future work in Section 7.

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

[1] Unexpected messages are MPI messages which arrive at a node before a matching receive is posted using an `MPI_Recv` or `MPI_Irecv`

2 Related Work

The ping-pong latency benchmark has become a standard by which all high-performance networks are evaluated. NetPIPE[19] and Netperf[1] are benchmarks that are commonly used to mesaure ping-pong latency, but it is almost as common for individuals (or network vendors) to write their own. With the advent of MPI, there has been surprisingly little published research on more realistic latency measurements. Our preliminary work in [17] presented a new latency micro-benchmark that includes the variance in transmission time, which the standard ping-pong benchmark does not reveal.

Overall, there have been some attempts at providing a more complete set of micro-benchmarks that better characterize the behavior of real applications and/or expose potential performance advantages that applications may leverage. Our work in [10] is a micro-benchmark suite that measures the potential for overlap that an MPI implementation offers. Recent work in [11] contains results that use micro-benchmarks to measure overhead, overlap potential, the impact of buffer re-use, and memory consumption. Other work has studied the impact of LogP[8] parameters on application performance[14] as well as the LogGP[2] parameters of modern networks[3]. All of these provide a more complete picture of the interaction of the application and the network. This paper differentiates itself by focusing on the impact of MPI queue usage scenarios that can occur in applications. The new benchmarks presented here illustrate how various queue usage scenarios can serve to increase the overhead (or gap, depending on where queue processing occurs) of the baseline hardware as defined by the LogP model.

3 Benchmarks

Studying the impacts of MPI queue length on message latency required two new benchmarks. Each is based on a standard ping-pong benchmark with some modifications. The first benchmark varies both the length of the pre-posted receive queue and the portion of that queue that is traversed. Ping-pong message latency is then measured in that context. The second benchmark varies the length of the unexpected queue and also measures the impact on message latency.

3.1 Pre-posted Receive Queue Impact

The benchmark designed to measure the impact of changes in the pre-posted receive queue length provides three degrees of freedom: the length of the pre-posted receive queue, the portion of the pre-posted receive queue that is traversed, and the size of the message. This enables the user to measure the impacts of both the receive queue length and the impact of actual queue traversal.

Pseudo-code describing the benchmark on each of two nodes is shown in Figure 1. Each node must first post the number of receives that are to be traversed when the actual latency measuring message arrives. These receives use tags that do not match the latency measuring message. This must occur on both nodes since the ping-pong latency will be divided by two to determine the one-way latency. Next, the receive that matches the latency message is posted, followed by the remaining queue entries that are requested (which will not be traversed when the latency message is received). Both nodes then enter a barrier operation that is designed to insure that node 1 exits first (this is not the standard `MPI_Barrier`) so that the barrier does not inadvertently interfere with the latency measurement. Following this, the standard ping-pong latency is measured and then the extra receives that were posted are cleared out of the queue (by sending a matching stream of messages).

Measurements for this paper use the average time from 1000 iterations of the core routine shown in Figure 1. The inner loop iterates 1000 times and sums the times from all of the iterations. The time is divided by 1000 and then divided by two to obtain a one-way latency measure.

3.2 Unexpected Message Queue Impact

The benchmark created to assess the impact of unexpected message queue length on message latency only allows the length of the unexpected message queue and the size of the message to be varied. It deviates from the traditional way of measuring latency in that it includes the time to post the receive for the latency measuring message as part of the latency. This better reflects the way that MPI is actually used by applications, which typically have some number of iterations and posts receives in each iteration.

Figure 2 shows pseudo-code for the benchmark that measures the impact of a long unexpected queue. The requested number of unexpected messages is first sent to each processor. The processors barrier in such a way that node 1 is guaranteed to exit first. Then, node 0 does a non-blocking send while node 1 waits on the message. The posting of the receive (and thus traversal of the unexpected queue) on node 0 is overlapped with this send. On node 1, a send is performed as soon as the receive completes. When the ping-pong latency timing is complete, the unexpected messages are cleared by posting matching receives. The inner loop is timed for 1000 iterations to obtain an average, and this value is divided by two to obtain the one-way latency.

This benchmark required a number of design decisions. First, node 1 exits the barrier first allowing some of the time traversing the unexpected queue to be hidden. This was an attempt to be as fair as possible in measuring the time. Oth-

```
prepost_traversed_receives();
post_latency_receive();
prepost_untraversed_receives();
barrier();
begin_timer();
send_message();
wait_for_response();
end_timer();
clear_receives();
```

(a)

```
prepost_traversed_receives();
post_latency_receive();
prepost_untraversed_receives();
barrier();
wait_for_message();
send_response();
clear_receives();
```

(b)

Figure 1. Pseudo-code for pre-posted queue impact benchmark: (a) node 0, and (b) node 1

```
send_unexpected_messages();
barrier();
begin_timer();
nonblocking_send_message();
post_latency_receive();
wait_for_response();
end_timer();
clear_unexpected_messages();
```

(a)

```
send_unexpected_messages();
barrier();
wait_for_message();
send_response();
clear_unexpected_messages();
```

(b)

Figure 2. Pseudo-code for unexpected queue impact benchmark: (a) node 0, and (b) node 1

erwise, the extra time for node 1 to exit the barrier would also be counted against the unexpected message behavior of the network. The second design choice made was to perform the non-blocking send on node 0 so that the post of the receive could be overlapped with it. Again, since this benchmark deviates from the standard ping-pong test by including the time to post a receive, a conservative choice was made to allow the network as much opportunity for overlap as possible.

4 Experimental Platforms

A number of platforms were evaluated using the newly developed micro-benchmarks. In the commodity space, Myrinet (Lanai-9), Quadrics (Elan3), and InfiniBand network hardware was evaluated. Myrinet (Lanai-X) and Quadrics (Elan4) hardware evaluations will be added as soon as hardware becomes available. To constrast the commodity networks, the custom network on ASCI Red was also evaluated. In addition to differences in hardware, differences in programming models were considered for both Quadrics and the ASCI Red network.

The Myrinet[4] Lanai9 evaluation platform contains dual processor, 2.4 GHz Pentium-4 nodes, but only one processor per node was used for testing. Each node has 2 GB of memory. The network used Lanai-9 (Myrinet 2000) adapters running GM[15]. The software used was MPICH-GM on RedHat 7.3 with a Linux 2.4.20 kernel. InfiniBand[9] testing was done using dual processor, 3.06 GHz Pentium-4 nodes, and again, only one processor per node was used for testing. The network uses Voltaire HCAs and switches to provide $4\times$ InfiniBand. The software stack consists of MVAPICH[12] on RedHat 9.0 with a Linux 2.4.22 kernel.

The Myrinet[4] LanaiX evaluation platform contains dual processor, 3.06 GHz Pentium-4 nodes, but only one processor per node was used for testing. Each node has 2 GB of memory. The network used Lanai-X adapters running GM[15]. The software used was MPICH-GM on SuSE Linux 9.0 with a Linux 2.4.25 kernel.

Quadrics[16] Elan3 hardware was tested on dual processor, 1 GHz Pentium-III nodes with 1 GB of memory. Elan3 network hardware was used with a RedHat 7.3 distribution and a Linux 2.4.20 kernel. Two versions of MPI software were used: the default MPICH variant from Quadrics using the TPorts API and a variant of MPICH 1.2.5 built at Sandia[5] using the Cray SHMEM API[7].

Quadrics Elan4 hardware was tested on dual processor, 2 GHz Opteron nodes with 2 GB of memory. Elan4 network hardware was used with a SuSe Linux Enterprise 8.0 distribution and a modified Linux 2.4.21 kernel. Two versions of MPI software were used: the default MPICH variant from Quadrics using the TPorts API and a variant of MPICH 1.2.5 built at Sandia[5] using the Cray SHMEM

API[7].

Compared to the commodity platforms discussed, ASCI Red[20] is a relatively unique system. It is a large-scale supercomputer comprised of more than 4500 dual-processor nodes connected by a high-performance custom network fabric. Each compute node has two 333 MHz Pentium II Xeon processors. Each compute node has a network interface, called a CNIC, that resides on the memory bus and allows for low-latency access to all of physical memory on a node. The CNIC interface connects each node to a 3-D mesh network that provides a 400 MB/s uni-directional wormhole-routed connection between the nodes. The CNIC interface is capable of sustaining the 400MB/s node-to-node transfer rate to and from main memory across the entire machine.

The software environment on ASCI Red is also significantly different from the standard commodity model. The compute nodes run Cougar, a variant of the Puma lightweight kernel that was designed and developed by Sandia and the University of New Mexico for maximizing both message passing throughput and application resource availability [18].

Cougar uses a simple network protocol built around the Portals message passing interface [18]. Portals are data structures in an application's address space that determine how the kernel should respond to message passing events. Portals allow the kernel to deliver messages directly from the network to the application's memory.

Cougar is not a traditional symmetric multi-processing operating system. Instead, it supports four different modes that allow different distributions of application processes on the processors. The following provides an overview of two of these processor modes that are relevant to this paper. More details can be found in [13].

The simplest processor usage mode is to run both the kernel and application process on the system processor. This mode (proc 0 mode) is commonly referred to as "heater mode" since the second processor is not used and only generates heat. In this mode, the kernel runs only when responding to network events or in response to a system call from the application process.

In the second mode, message co-processor mode (or proc 1 mode), the kernel runs on the system processor and the application process runs on the user processor. When the processors are configured in this mode, the kernel runs continuously waiting to process events from external devices or service system call requests from the application process. Because the time to transition from user mode to supervisor mode and back can be significant, this mode offers the advantage of reduced network latency and faster system call response time.

5 Results

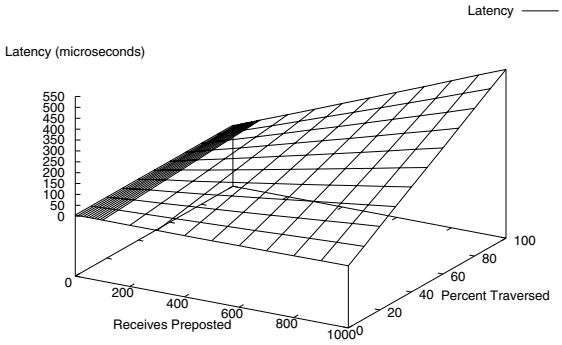
Each benchmark was used to measure each of the systems in question. These measurements highlight differences in network hardware and system integration. They also highlight distinct differences between the relative complexities of the communication APIs that are used by the MPI implementation on the networks by measuring more than one API on a number of the platforms.

5.1 Pre-posted Queue Impacts

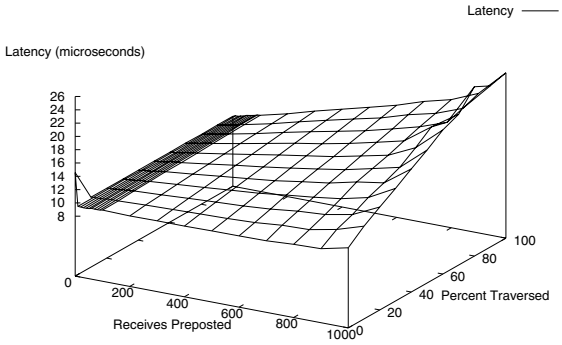
The length of the pre-posted receive queue has two distinct impacts on the latency of messages. First, a long pre-posted receive queue has implications for resource usage and resource management. Thus, even if most incoming messages match the first entry, a long posted receive queue can have a negative impact on message performance. Second, each item traversed in the pre-posted queue takes many processor cycles. This can significantly increase the time to handle an incoming message. Graphs in this section show message latency as the number of receives pre-posted is varied from 0 to 1000 and the percent of that queue traversed is varied from 0 to 100 to show both of these effects.

Figures 3(f) and 4(e) and (f) show the impact of the length of the pre-posted receive queue on InfiniBand, Myrinet Lanai9, and Myrinet LanaiX, respectively. Even with a 2.4 GHz (or greater) Pentium-4 handling the pre-posted receive queue traversal, the latency impact of a long queue is quite noticeable. Simply having a long queue has no impact on latency; however, latency can be increased by as much as 60% when a large fraction of that queue is traversed. These increases in latency manifest themselves as an increase in overhead (from the LogP model) when queue traversal is handled on the host. This indicates a need for a better data structure than a simple linear list, even when a fast processor is handling the list traversal.

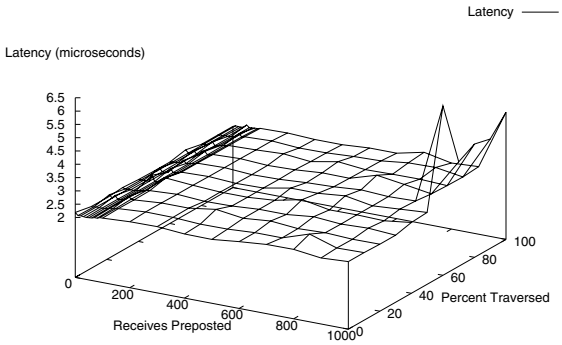
Figure 3(a) and (b) compare the impact of the length of the pre-posted receive queue on Quadrics Elan3 using the TPorts and SHMEM APIs. This provides a direct comparison between APIs that offload many of the MPI semantics and APIs that do not on a single hardware platform. The impact of the pre-posted receive queue on MPI over SHMEM on Quadrics Elan3 (Figure 3(b)) is comparable to the behavior of MPI on InfiniBand and Myrinet (when adjusted for differences in host processor performance). The impact on MPI over the TPorts API, however, is much more drastic. At the baseline, increasing the length of the pre-posted receive queue to 1000 entries (even if the first entry matches the incoming message) doubles the latency of messages. Since TPorts offloads this queue onto the network interface, this is likely due to a resource management issue. As more items in that queue must be traversed to find



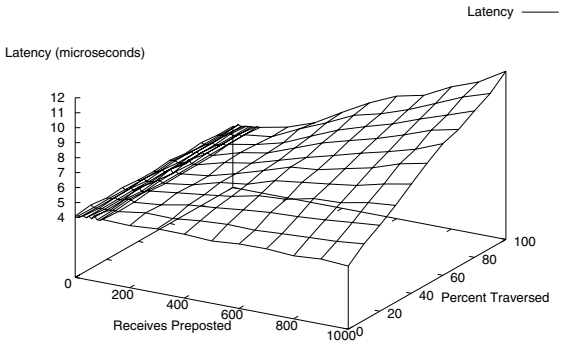
(a) Quadrics Elan3 using TPorts



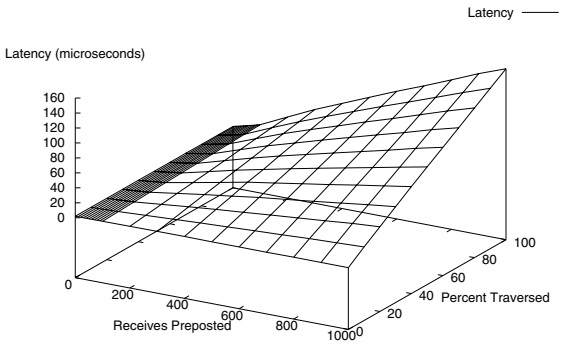
(b) Quadrics Elan3 using SHMEM



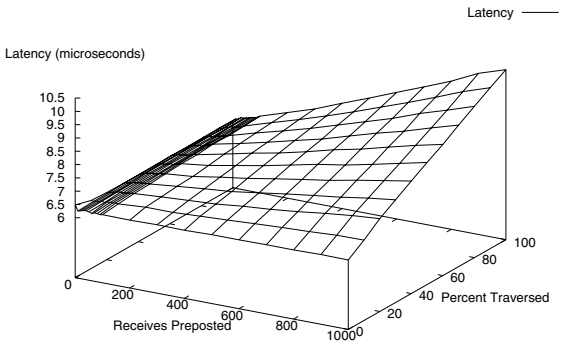
(c) Quadrics Elan4 using TPorts



(d) Quadrics Elan4 using SHMEM



(e) Quadrics Elan4 using TPorts (no hashing)



(f) InfiniBand using VAPI

Figure 3. Latency impacts as the pre-posted receive queue is varied

a match, we see the impacts of the slow network interface processor. Each item in the queue that is traversed adds approximately 0.49 μs to the latency of the message. These are effectively increases in the gap because the queue traversal is handled on the NIC. Results from the newest Quadrics hardware (Elan4, Figures 3(c) and (d)) show a completely different picture. The TPorts interface appears to be much faster than the SHMEM interface, even in the presence of long queues. This inconsistency was traced to an apparent addition of a hashing algorithm based on the MPI envelope information. Since different MPI tags were used to select the message, the hash algorithm effectively prevented traversing the entire queue; however, a simple change to the benchmark (wildcarding the source address, which is a relatively common application behavior) produced the graph in Figures 3(e). While the performance of the embedded microprocessor has greatly improved (dropping the penalty to approximately 0.1 μs per queue element for small queues and 0.15 μs per queue element for long queues), it still shows a significant performance degradation when the entire list must be traversed.

Figure 4(a) and (b) compare the Portals and SHMEM communication APIs on ASCI Red in Proc 0 mode. In Proc 0 mode, all of the communication API processing is handled on the application processor. This comparison highlights the impact of the complexity of the pre-posted receive queue list traversal on message latency when the length of that queue is varied. Portals, which has semantics that are designed to handle MPI as well as other supercomputer message traffic such as I/O, provides a rich set of queue traversal and matching semantics. These matching semantics cannot be disabled; thus, each match list item that must be traversed adds significant overhead. For SHMEM, the semantics are very simple and MPI must implement all of the queue traversal and matching semantics. This allows MPI to customize the queue traversal and matching operations to match its needs. The penalty is that features such as independent progress and matching offload are lost. The ultimate result is that Portals has significantly lower latency when a small number of queue items are traversed. When a large number of queue items are traversed, the robustness of the portals matching semantics causes it to have a higher overall latency.

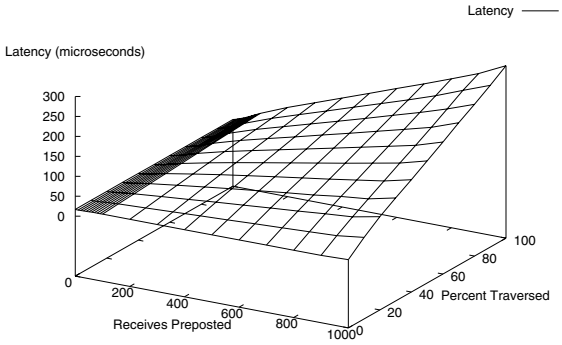
Comparing Figure 4(c) and (d) to Figure 4(a) and (b) illustrates the impact of Proc 1 mode. Proc 1 mode offloads the communication API onto a second processor in an SMP configuration. Thus, all application (and MPI) processing occur on one processor in a node while all Portals or SHMEM processing occur on a second, equivalent processor in the same node. When MPI uses the SHMEM API, it sees a consistent benefit from this offloading ability. In contrast, when MPI uses the Portals API, it sees a significant benefit when a small number of elements are traversed in the posted receive queue, but pays an extremely large penalty when a large number of posted receive queue elements are traversed. This is clearly a performance bug in the software that handles offloading for Portals; however, it points out the complexities in properly handling such offloading.

5.2 Unexpected Message Queue Impacts

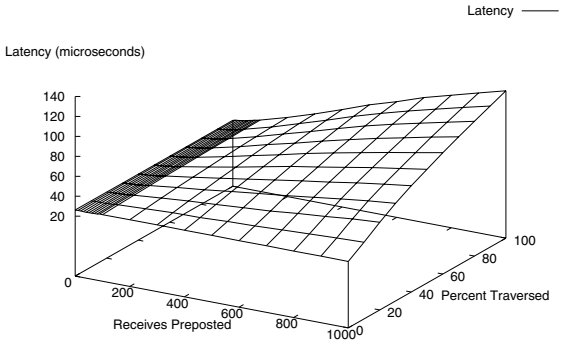
Unexpected message queue length impacts latency in a slightly different way than the pre-posted queue length. The entire unexpected message queue must be traversed each time a receive is posted, whereas the pre-posted queue only needs to be traversed until a match is found. Another difference is that the posting of a receive can be overlapped with a send (and therefore partially hidden by the latency of the physical layer) if non-blocking operations are used for both (and the send is initiated first).

Figure 5(a) shows the increase in latency as the length of the unexpected message queue is increased for Myrinet and InfiniBand. Note that Myrinet is relatively unaffected by the length of the unexpected message queue. This is because, as with the pre-posted queue traversal, all of the "work" in MPI is performed on the host processor. For these tests, the host processor is a multi-gigahertz Intel Pentium-4. Such a processor is capable of traversing the unexpected message queue quickly; thus, virtually all of the added latency is hidden in the message transfer time. InfiniBand behaved similarly, but, unfortunately, the InfiniBand results were cut short at 100 unexpected messages due to an apparent bug in the MPI library. Although the queue traversal times are hidden from these latency tests, they are serving to increase the overhead (from the LogP model) of the communications beyond that required by the baseline hardware.

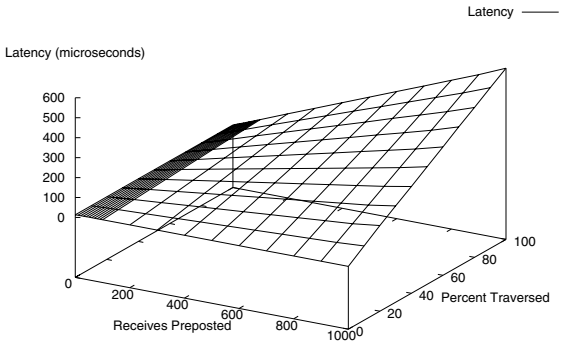
Figure 5(b) shows the increase in latency as the length of the unexpected message queue is increased for Quadrics Elan3 and the custom network on ASCI Red. For Quadrics Elan3 using MPI over the TPorts API, latency increases by approximately 0.1 μs for each additional item in the unexpected queue (with large numbers of unexpected messages — at small numbers of unexpected messages, much of this increase is hidden by message transfer time). This is because Quadrics offloads unexpected message handling and must traverse the unexpected queue with the relatively slow embedded processor on the card each time a receive is posted. This leads to an increase in the LogP "gap" while the card is busy processing the incoming message. The curve for Quadrics using MPI over the SHMEM API shows a striking difference from the TPorts results. When unexpected queue processing occurs on the host, relatively little impact is seen on the latency — even with 1000 unexpected messages in the unexpected message queue. This is simply a matter of using a faster processor with a larger



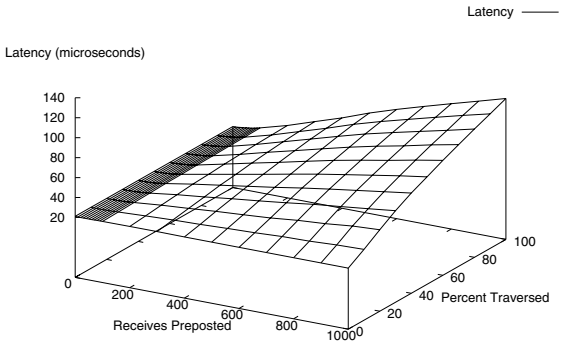
(a) ASCI Red/Portals/Proc0



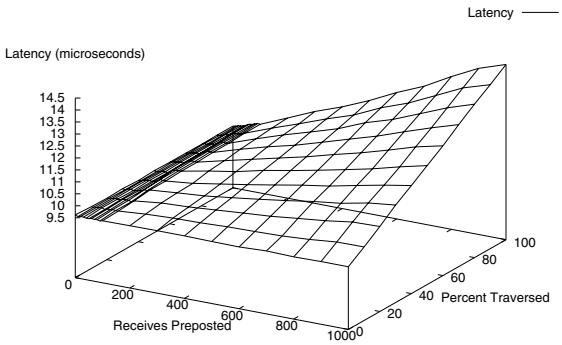
(b) ASCI Red/SHMEM/Proc0



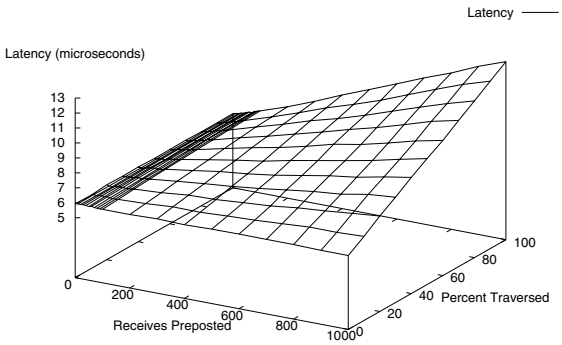
(c) ASCI Red/Portals/Proc1



(d) ASCI Red/SHMEM/Proc1



(e) Myrinet Lanai9 GM



(f) Myrinet LanaiX GM

Figure 4. Latency impacts as the pre-posted receive queue is varied

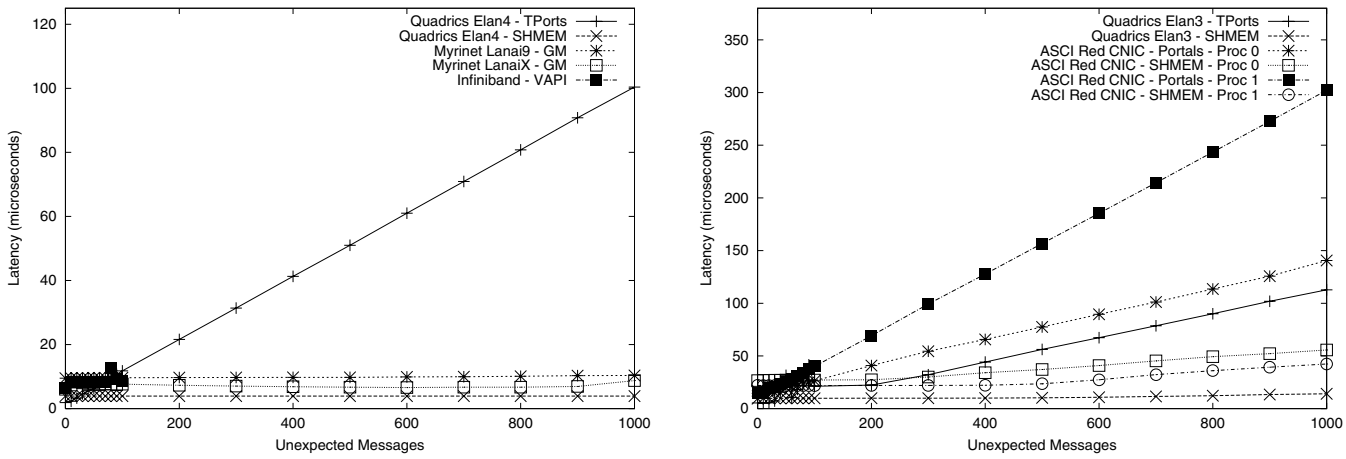


Figure 5. Latency impacts as the unexpected queue is varied

cache to perform the exact same list processing. This behavior changes little when moving to Quadrics Elan4 (Figure 5(a)); however, minimum latencies were used for the Quadrics Elan4 TPorts data due to unexplained variability in our current hardware platform. This is simply an issue with a new system that has not been fully brought up yet.

Results from ASCI Red, shown in Figure 5(b), explore issues of both protocol complexity and network integration. ASCI Red makes available two communication APIs — Portals, which provides a rich set of semantics including much of the queue processing needed by MPI, and the lightweight Cray SHMEM interface. Comparing Portals performance to SHMEM performance, it is clear that the rich semantics of Portals come at a cost. Although Portals performs better when there is a small number of unexpected messages, the robust matching semantics provided by Portals cause it to yield higher latency when there is a large number of unexpected messages. The second interesting feature of this graph is the comparison of Proc 0 and Proc 1 modes. When offloading the communications API to a second (equally fast) processor in an SMP node (Proc 1 mode), both Portals and SHMEM have better performance with small numbers (under 100) of unexpected messages. As the number of unexpected messages grows large, however, Portals (which offloads matching to the second processor) ramps up to significantly higher latencies than Proc 0 mode while SHMEM (which performs queue processing on the application processor) continues to see an advantage from Proc 1 mode. For Portals, there seems to be a detrimental impact from the interaction of the two processors for longer unexpected message queues. While this is clearly a performance bug in Portals offloading software, it highlights an issue that is easy to implement badly in offloading scenarios.

6 Conclusions

Two new benchmarks were introduced to evaluate MPI latency in more realistic usage scenarios. These benchmarks highlight the key weakness in networks that offload protocol processing onto the network interface: the network interface processor is typically much slower than the host processor. Thus, under some usage scenarios, they have much poorer performance than competing networks that use the host processor for these tasks. Specifically, InfiniBand and Myrinet perform as much as an order of magnitude better than Quadrics when MPI queues are lengthy. Similarly, the weight of the protocol underlying MPI can significantly increase latency when longer MPI queues occur. When combined with application analysis[6] that indicates that longer queues sometimes occur, this suggests that more processing power needs to be allocated to network functions.

7 Future Work

This work is part of a broader overall effort to characterize applications and to develop successful benchmarking techniques. Future efforts will include further analysis of application codes to explore how MPI is used. In addition, these benchmarks will be enhanced and other benchmarks will be designed to test systems under typical usage scenarios. New benchmark development efforts will include a focus on key network features such as measuring the ability to overlap communications and computation. In addition, network bandwidth under typical loads (e.g. receiving

from multiple simultaneous sources) will be tested. Finally, the benchmarking of collective operations will be investigated. Collective benchmarks are particularly unrealistic in that they measure the time to perform a large number of consecutive operations. This is yet another benchmarking scenario that never occurs in practice.

References

[1] *Netperf*. http://www.netperf.org.

[2] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Sheiman. LogGP: Incorporating long messages into the LogP model. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[3] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick. An evaluation of current high-performance networks. In *17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, Apr. 2003.

[4] N. J. Boden, D. Cohen, R. E. F. A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[5] R. Brightwell. A new MPI implementation for Cray SHMEM. Technical report, Sandia National Laboratories. Work in progress.

[6] R. Brightwell and K. D. Underwood. An analysis of NIC resource usage for offloading MPI. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters*, Santa Fe, NM, April 2004.

[7] Cray Research, Inc. *SHMEM Technical Note for C, SG-2516 2.3*, October 1994.

[8] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel computation. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1993.

[9] Infiniband Trade Association. *http://www.infinibandta.org*, 1999.

[10] W. Lawry, C. Wilson, A. B. Maccabe, and R. Brightwell. COMB: A portable benchmark suite for assessing MPI overlap. In *IEEE International Conference on Cluster Computing*, September 2002. Poster paper.

[11] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. K. Panda. Performance comparison of MPI implementations over InfiniBand, Myrinet and Quadrics. In *The International Conference for High Performance Computing and Communications (SC2003)*, November 2003.

[12] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda. High performance RDMA-based MPI implementation over InfiniBand. In *Proceedings of the 2003 International Conference on Supercomputing (ICS-03)*, pages 295–304, New York, June 23–26 2003. ACM Press.

[13] A. B. Maccabe, R. Riesen, and D. W. van Dresser. Dynamic processor modes in Puma. *Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS)*, 8(2):4–12, 1996.

[14] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[15] Myricom, Inc. The GM Message Passing System. Technical report, Myricom, Inc., 1997.

[16] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, January/February 2002.

[17] R. Riesen, R. Brightwell, and A. B. Maccabe. Measuring MPI latency variance. In J. Dongarra, D. Laforenza, and S. Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 10th European PVM/MPI Users' Group Meeting, Venice, Italy, September/October 2003 Proceedings*, volume 2840 of *Lecture Notes in Computer Science*, pages 112–116. Springer-Verlag, 2003.

[18] L. Shuler, C. Jong, R. Riesen, D. van Dresser, A. B. Maccabe, L. A. Fisk, and T. M. Stallcup. The Puma operating system for massively parallel computers. In *Proceeding of the 1995 Intel Supercomputer User's Group Conference*. Intel Supercomputer User's Group, 1995.

[19] Q. Snell, A. Mikler, and J. Gustafson. NetPIPE: A Network Protocol Independent Performance Evaluator . In *Proceedings of the IASTED International Conference on Intelligent Information Management and Systems*, June 1996.

[20] S. R. W. Timothy G. Mattson, David Scott. A TeraFLOPS Supercomputer in 1996: The ASCI TFLOP System. In *Proceedings of the 1996 International Parallel Processing Symposium*, 1996.